Contents

Data Compression
Lecture Notes

Dr. Faisal Aslam

1: Lecture 1: Introduction to Data Compression

1.1 Learning Objectives

By the end of this lecture, students will be able to:

- Understand the motivation and benefits of data compression
- Differentiate between lossless and lossy compression techniques
- Compute and interpret common compression performance metrics
- Apply the Huffman coding algorithm step by step
- Analyze real-world compression trade-offs

1.2 Introduction and Motivation: Why Compress Data?

Data compression is the process of representing information using fewer bits than its original form. It is a fundamental component of modern computing systems, enabling efficient storage, faster communication, and reduced operational costs.

Everyday applications of compression include:

- Streaming audio and video
- Image storage and sharing
- File archiving and backups
- Network communication and cloud services

Definition

Data Compression is the process of reducing the number of bits required to represent information, either:

- **Losslessly**: allowing exact reconstruction of the original data
- **Lossily**: allowing controlled loss of information to achieve higher compression

1.2.1 Benefits of Data Compression

Data compression provides three key benefits that are critical in modern computing:

1. **Reduce Storage Space**:
   - Allows more data to be stored in the same physical space
   - Enables archival of historical data that would otherwise be discarded
   - Reduces hardware requirements for storage systems
2. **Reduce Communication Time and Bandwidth**:
   - Enables faster file transfers and downloads
   - Makes high-quality streaming (4K/8K video) practical over limited bandwidth
   - Reduces latency in real-time applications like video conferencing and online gaming
   - Allows IoT devices to transmit data efficiently over wireless networks
3. **Save Money**:
   - Reduces cloud hosting costs (storage and egress fees)
   - Lowers communication costs for data transmission
   - Decreases capital expenditure on storage hardware
   - Reduces energy consumption for data centers and network infrastructure

1.3 Lossless vs. Lossy Compression

1.3.1 Lossless Compression

Lossless compression guarantees perfect reconstruction of the original data. It is essential when accuracy and data integrity are critical.

Typical applications:

- Text files and source code
- Executables and databases
- Medical, scientific, and legal data

1.3.2 Lossy Compression

Lossy compression achieves higher compression ratios by discarding information that is less perceptible or less important.

Typical applications:

- Audio (MP3, AAC)
- Images (JPEG)
- Video (H.264, H.265)

1.3.3 Choosing Between Lossless and Lossy

Factor	**Lossless Compression**	**Lossy Compression**
Reconstruction	Exact	Approximate
Data sensitivity	High	Moderate to low
Typical ratios	Low to moderate	High
Quality impact	None	Controlled degradation

Table 1: Lossless vs. Lossy Compression

1.4 Compression Performance Metrics

1.4.1 Size-Based Metrics

$$\text{Compression Ratio (CR)} = \frac{\text{Original Size}}{\text{Compressed Size}}$$
$$\text{Compression Factor} = \frac{\text{Compressed Size}}{\text{Original Size}}$$
$$\text{Space Savings (\%)} = \left(1 - \frac{\text{Compressed Size}}{\text{Original Size}}\right) \times 100\%$$

Interpretation:

- Larger compression ratios indicate better compression
- Smaller compression factors indicate better compression

1.4.2 Rate-Based Metrics

$$\text{Bits per Sample (bps)} = \frac{\text{Compressed Size (bits)}}{\text{Number of samples}}$$
$$\text{Bit-rate (bps)} = \frac{\text{Compressed Size (bits)}}{\text{Time (seconds)}}$$

These metrics are particularly important in audio and video compression systems.

1.5 Worked Example: Audio Compression Metrics

Example

Uncompressed Audio Properties

- Duration: 180 seconds
- Sampling rate: 44.1 kHz
- Bit depth: 16 bits
- Channels: 2 (stereo)

Original Size Calculation

$$\begin{aligned}\text{Total samples} &= 180 \times 44{,}100 \times 2 = 15{,}876{,}000\\ \text{Size (bits)} &= 15{,}876{,}000 \times 16 = 254{,}016{,}000\\ \text{Size (MB)} &= \frac{254{,}016{,}000}{8 \times 1{,}048{,}576} \approx 30.27\end{aligned}$$

Compression Results

Method	Size (MB)	CR	Savings	Bit-rate
FLAC (lossless)	18.16	1.67:1	40%	807 kbps
MP3 @ 320 kbps	6.75	4.49:1	77.7%	320 kbps
AAC @ 256 kbps	5.40	5.61:1	82.2%	256 kbps

1.6 Huffman Coding

Huffman coding is a widely used **lossless compression algorithm** that assigns variable-length binary codes to symbols based on their frequencies. More frequent symbols receive shorter codes.

1.6.1 Step-by-Step Huffman Coding Example

Example

Message: `MISSISSIPPI RIVER` (17 characters including space)

Symbol Frequencies

Symbol	Frequency
I	5
S	4
P	2
R	2
M	1
V	1
E	1
(space)	1

Tree Construction

1. Combine M(1) + V(1) $\rightarrow$ 2
2. Combine E(1) + (space)(1) $\rightarrow$ 2
3. Combine P(2) + R(2) $\rightarrow$ 4
4. Combine 2 + 2 $\rightarrow$ 4
5. Combine 4 + 4 $\rightarrow$ 8
6. Combine I(5) + S(4) $\rightarrow$ 9
7. Combine 8 + 9 $\rightarrow$ 17

One Possible Code Assignment

Symbol	Code	Length
I	00	2
S	01	2
P	100	3
R	101	3
M	1100	4
V	1101	4
E	1110	4
(space)	1111	4

Compressed Size

$$5(2) + 4(2) + 2(3) + 2(3) + 4(1) = 52 \text{ bits}$$

Original Size (ASCII) $= 17 \times 8 = 136$ bits
Compression Ratio $= 136/52 \approx 2.62 : 1$

1.6.2 Key Properties of Huffman Coding

- Produces prefix-free codes
- Enables instantaneous decoding
- Guarantees minimum average code length among prefix codes
- Widely used in practical compression systems

1.7 End of Chapter Questions

Exercise 1.0

Problem 1: Basic Compression Metrics

An uncompressed grayscale image has the following properties:

- Resolution: 1024×1024 pixels
- Bit depth: 8 bits per pixel

After compression, the image size is 320 KB.
Calculate:

(a) Original image size in KB

(b) Compression ratio

(c) Compression factor

(d) Space savings percentage

Exercise 1.1

Problem 2: Audio Bit-rate and Storage

A mono audio recording has the following parameters:

- Duration: 5 minutes
- Sampling rate: 48 kHz
- Bit depth: 16 bits

The file is compressed using a lossy codec to a constant bit-rate of 192 kbps.
Calculate:

(a) Size of the uncompressed audio file in MB

(b) Size of the compressed file in MB

(c) Compression ratio

(d) Bits per sample after compression

Exercise 1.2

Problem 3: Comparing Compression Options

A video clip has an uncompressed data rate of 120 Mbps. Three compression options are available:

Option	Compressed Bit-rate
A	6 Mbps
B	3 Mbps
C	1.5 Mbps

For each option, calculate:

(a) Compression ratio

(b) Data consumed for a 10-minute video (in MB)

Which option would you choose for:

(i) Live video streaming?

(ii) Archival storage?

Briefly justify your answers.

Exercise 1.3

Problem 4: Huffman Coding Construction

Given the following symbol frequencies:

Symbol	Frequency
A	10
B	8
C	6
D	5
E	4
F	3
G	2
H	2

(a) Construct the Huffman tree step by step

(b) Assign a binary code to each symbol

(c) Compute the total number of bits required to encode the message

(d) Calculate the average number of bits per symbol

Exercise 1.4

Problem 5: Fixed-Length vs. Huffman Coding
Using the symbol set from Problem 4:

(a) Determine the minimum fixed-length code required

(b) Compute the total number of bits using fixed-length coding

(c) Compare the result with Huffman coding

(d) Calculate the percentage reduction in total bits achieved by Huffman coding

Exercise 1.5

Problem 6: Text Compression Scenario
A text file contains 50,000 characters and is stored using 8-bit ASCII encoding. After compression using a lossless algorithm, the file size becomes 18 KB. Calculate:

(a) Original file size in KB

(b) Compression ratio

(c) Compression factor

(d) Space savings percentage

Explain why compression ratios for text files vary significantly depending on content.

Exercise 1.6

Problem 7: Practical Design Question
You are designing a compression system for a wearable health-monitoring device that:

- Records sensor data continuously
- Has limited storage capacity
- Requires exact data reconstruction
- Operates on a low-power processor

(a) Should the system use lossless or lossy compression? Explain.

(b) Which performance metrics are most important in this scenario?

(c) Would a variable-length coding scheme be appropriate? Why or why not?

2: Lecture 2: Theory of Compression — Limits and Optimality

2.1 Types of Codes: From Ambiguous to Instantaneous

Definition

Types of Codes

- **Non-singular Code**: Each source symbol maps to a distinct codeword

$$x_i \neq x_j \Rightarrow C(x_i) \neq C(x_j)$$

- **Uniquely Decodable Code**: Every finite sequence of codewords corresponds to exactly one sequence of source symbols

$$C(x_1)C(x_2)\cdots C(x_n) = C(y_1)C(y_2)\cdots C(y_m) \Rightarrow n = m \text{ and } x_i = y_i$$

- **Prefix Code (Instantaneous Code)**: No codeword is a prefix of another codeword

$$\forall i \neq j : \quad C(x_i) \text{ is not a prefix of } C(x_j)$$

Key Relationships:

$$\text{Prefix Codes} \subset \text{Uniquely Decodable Codes} \subset \text{Non-singular Codes}$$

Important

Why Prefix Codes are Special

- **Instantaneous decoding**: Can decode as soon as codeword ends (no lookahead needed)
- **Tree representation**: Always correspond to leaves of a binary tree
- **Kraft inequality**: Always satisfy $\sum 2^{-\ell_i} \leq 1$
- **Practical**: Used in Huffman coding, many real-world compressors

Example

Example: Comparing Different Code Types

For symbols $\{A, B, C, D\}$ with probabilities $\{0.5, 0.25, 0.125, 0.125\}$:

Code Type	A	B	C	D	Property
Non-singular	0	1	00	11	Distinct but ambiguous: "00" = AA or C?
Uniquely decodable	0	01	011	0111	Unique but need lookahead
Prefix code	0	10	110	111	Instant decoding: "0" = A, stop
Optimal prefix	0	10	110	111	Also Huffman optimal

Decoding examples:

- **Prefix code "010110"**: 0→A, 10→B, 110→C = "ABC" (instant)
- **Uniquely decodable "00111"**: Need to scan ahead to determine split
- **Non-singular "00"**: Ambiguous! Could be "AA" or "C"

Key insight: Prefix codes sacrifice some flexibility in codeword lengths (must satisfy Kraft inequality) for the benefit of instantaneous decoding.

2.2 Basic Terminology and Notation

Definition

Alphabet

An *alphabet* $\mathcal{X}$ is a finite set of possible symbols. Examples:

- Binary alphabet: $\mathcal{X} = \{0, 1\}$
- English letters: $\mathcal{X} = \{\texttt{A}, \ldots, \texttt{Z}\}$
- Bytes: $\mathcal{X} = \{0, 1, \ldots, 255\}$

Definition

Symbol

A *symbol* is a single element drawn from an alphabet. For example, the letter `E` is a symbol from the English alphabet.

Definition

Random Variable

A *random variable* X is a function that assigns a symbol or value to each outcome in a sample space:

$$X : \Omega \to \mathcal{X}$$

- Ω: Sample space (e.g., all possible states of a data source)

- $\mathcal{X}$: Set of possible values (alphabet, e.g., $\{0, 1\}$, ASCII characters)
- For each $\omega \in \Omega$, $X(\omega)$ is the value assigned to outcome ω

Example: Binary Source

- $\Omega = \{\text{emits } 0, \text{emits } 1\}$ (or could be more complex underlying physics)
- $\mathcal{X} = \{0, 1\}$
- $X(\text{emits } 0) = 0$, $X(\text{emits } 1) = 1$
- Probabilities: $P(X = 0) = P(\{\omega : X(\omega) = 0\}) = p$, $P(X = 1) = 1 - p$

Why this matters for compression:

- The entropy $H(X)$ depends on the probability distribution induced by X
- For $x \in \mathcal{X}$: $P(X = x) = P(\{\omega \in \Omega : X(\omega) = x\})$
- $H(X) = -\sum_{x\in\mathcal{X}} P(X = x) \log_2 P(X = x)$

Definition

Source

A *source* is a process that generates a sequence of symbols $(X_1, X_2, X_3, \dots)$ according to some probability law. In this lecture, we assume discrete sources unless stated otherwise.

Definition

Message (or Sequence)

A *message* is a finite sequence of symbols generated by the source:

$$x^n = (x_1, x_2, \dots, x_n)$$

Compression algorithms operate on messages, not on individual symbols.

Definition

Code and Codewords

A *code* assigns a binary string (codeword) to each symbol or message.

- Source symbols $\to$ codewords (e.g., Huffman coding)
- Messages $\to$ bitstreams (e.g., arithmetic coding)

Definition

Block Length

The *block length* n is the number of source symbols grouped together and encoded as a unit. Larger block lengths generally allow better compression but increase delay and complexity.

Definition

Model

A *model* estimates the probabilities of symbols or sequences. Better models lead to better compression by reducing uncertainty.

2.3 Information and Redundancy: The Core Concepts

2.3.1 Information: A Formal Measure of Uncertainty Reduction

In information theory, information is defined rigorously as a **quantitative measure of the reduction in uncertainty** that results from observing the outcome of a random event.

Definition 2.1. *Let X be a random event that occurs with probability $p = \Pr(X)$. The **information content** (or* self-information*) $I(X)$ provided by the occurrence of X is defined as:*

$$I(X) = \log_b \left(\frac{1}{p}\right) = -\log_b(p)$$

where:

- $b = 2$ *yields **bits** (binary digits)*
- $b = e$ *yields **nats** (natural units)*
- $b = 10$ *yields **hartleys** or **dits***

Example

Predictability vs. Information:

- In a city where it rains every day, the statement "It rained today" conveys almost no information because it was expected
- A file that contains only the bit '1' provides very little information
- A coin that always lands heads produces outcomes, but no information

  Key idea: Perfect predictability implies zero information gain.

Example

Daily Weather Forecast — Information Content:

- Sunny in Phoenix (probability 0.9): $I = -\log_2 0.9 \approx 0.15$ bits
- Snow in Phoenix (probability 0.001): $I = -\log_2 0.001 \approx 9.97$ bits
- Rain in Seattle (probability 0.3): $I = -\log_2 0.3 \approx 1.74$ bits

Interpretation: Rare events carry more information because they reduce uncertainty the most.

2.3.2 Redundancy: The Enemy of Information and the Friend of Compression

Redundancy refers to predictable or repeated structure in data. It is what allows data to be represented using fewer bits.

1. **Spatial Redundancy**: Neighboring data values are highly correlated

   Example

   In a photograph of a clear blue sky, most neighboring pixels have nearly identical color values.

   - **Naive**: Store the RGB value of each pixel independently
   - **Smarter**: Encode repeated pixel values using run-length encoding
   - **Even smarter**: Predict each pixel from its neighbors and encode only the small prediction error

2. **Statistical Redundancy**: Some symbols occur far more frequently than others

   Example

   English letter frequencies:

Letter	Frequency	Letter	Frequency
E	12.7%	Z	0.07%
T	9.1%	Q	0.10%
A	8.2%	J	0.15%

   Frequent letters get shorter codes in variable-length coding schemes.

3. **Knowledge Redundancy**: Information already known to both encoder and decoder

4. **Perceptual Redundancy**: Information that humans cannot perceive

2.4 Entropy: The Fundamental Limit

2.4.1 What is Entropy? Different Perspectives

Definition

Shannon Entropy of a discrete random variable X with possible values $\{x_1, x_2, \ldots, x_n\}$ having probabilities $\{p_1, p_2, \ldots, p_n\}$:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i \quad \text{bits}$$

Two Complementary Interpretations:

1. **Average Information Content**: Expected value of information content across all symbols
2. **Uncertainty or Surprise**: Measures how uncertain we are about the next symbol

2.4.2 Calculating Entropy: Step by Step

Example

Binary Source Example - Detailed Calculation:
Consider a biased coin: P(Heads) = 0.8, P(Tails) = 0.2

Step 1: **Calculate individual information content**:

$$I_H = -\log_2(0.8) \approx 0.3219 \text{ bits}$$
$$I_T = -\log_2(0.2) \approx 2.3219 \text{ bits}$$

Step 2: **Calculate entropy as expected value**:

$$H = 0.8 \times 0.3219 + 0.2 \times 2.3219 = 0.7219 \text{ bits}$$

Step 3: **Verify using direct formula**:

$$H = -[0.8 \log_2(0.8) + 0.2 \log_2(0.2)] \approx 0.7219 \text{ bits}$$

Key Insights:

- **Extreme cases**:

- Fair coin (P=0.5): $H = 1.0$ bit (maximum uncertainty)
- Always heads (P=1.0): $H = 0$ bits (no uncertainty)
- 90% heads: $H \approx 0.469$ bits

2.4.3 Entropy of English Text: A Practical Case Study

Example

Calculating English Letter Entropy:
Based on letter frequencies in typical English text:

$$H \approx 4.18 \text{ bits/letter}$$

Layered Interpretation:

- **First-order entropy (letters independent)**: 4.18 bits/letter
- **Actual uncertainty is lower**: Letters have dependencies (Q→U)
- **Comparison with encoding schemes**:

Encoding Method	Bits/Letter
Naive (5 bits for 26 letters)	5.00
Huffman (letter-based)	4.30
Using digram frequencies	3.90
Using word frequencies	2.30
Optimal with full context	∼1.50

2.4.4 Beyond First-Order Entropy: The Full Picture

Higher-Order Entropies quantify uncertainty while accounting for increasing context:

- **Zero-order entropy (H_0)**: $H_0 = \log_2 |\mathcal{X}|$
- **First-order entropy (H_1)**: $H_1 = -\sum p(x) \log_2 p(x)$
- **Second-order entropy (H_2)**: $H_2 = -\sum p(x, y) \log_2 p(x|y)$
- **Nth-order entropy (H_N)**: $H_N = -\sum p(x_1, \ldots, x_N) \log_2 p(x_N \mid x_1, \ldots, x_{N-1})$

2.4.5 Entropy Rate

The **entropy rate** of a source is defined as the limiting uncertainty per symbol when arbitrarily long contexts are available:

$$H_\infty = \lim_{N\to\infty} H_N$$

2.4.6 The Entropy Theorem: Why It Matters

Definition

Expected Code Length

For a source with symbols $\{x_1, x_2, \ldots, x_n\}$ having probabilities $\{p_1, p_2, \ldots, p_n\}$, and a code that assigns codeword lengths $\{\ell_1, \ell_2, \ldots, \ell_n\}$, the **expected code length** L is:

$$L = \mathbb{E}[\ell(X)] = \sum_{i=1}^{n} p_i \ell_i \quad \text{(bits per symbol)}$$

This measures the average number of bits needed to encode one symbol from the source.

Definition

Compression Ratio and Efficiency

For a source with entropy $H(X)$ and code with expected length L:

- **Compression ratio**: $\rho = \frac{\text{original bits}}{\text{compressed bits}}$
- **Efficiency**: $\eta = \frac{H(X)}{L} \leq 1$
- **Redundancy**: $R = L - H(X) \geq 0$

Perfect compression occurs when $\eta = 1$ (100% efficient) and $R = 0$.

Important

Shannon's Source Coding Theorem (1948)

For a discrete memoryless source with entropy H and any $\epsilon > 0$:

1. **Converse (Impossibility Result)**:
   No lossless coding scheme can achieve expected code length $L < H$.

$$\boxed{L \geq H \quad \text{for any uniquely decodable code}}$$

2. **Achievability (Possibility Result)**:
   There exists a lossless coding scheme (specifically, block coding with suffi-

ciently large block size n) such that:

$$\boxed{H \leq L < H + \epsilon}$$

Equivalently: For any $\epsilon > 0$, $\exists n$ such that:

$$\frac{L_n}{n} < H + \epsilon$$

where L_n is the expected length for blocks of size n.

Interpretation:

- **Entropy is the fundamental limit**: H bits/symbol is the best we can ever do
- **We can get arbitrarily close**: With clever coding, we can approach this limit as closely as desired
- **The gap is achievable**: The "$+\epsilon$" represents practical overhead that can be made arbitrarily small

Example

Understanding the Theorem with Numbers

Consider a binary source with $p(0) = 0.9$, $p(1) = 0.1$:

$$H = -0.9 \log_2 0.9 - 0.1 \log_2 0.1 \approx 0.469 \text{ bits/symbol}$$

- **Naive coding**: Use 1 bit per symbol $\rightarrow L = 1.0$, efficiency $\eta = 0.469/1.0 = 46.9\%$
- **Huffman coding**: 0→0, 1→1 (same as naive!) $\rightarrow L = 1.0$, $\eta = 46.9\%$ *Why so bad?* Because we're coding symbols individually.
- **Block coding (n=2)**: Code pairs of symbols:

$$\begin{aligned}
00 &\rightarrow 0 \quad (\ell = 1, p = 0.81) \\
01 &\rightarrow 10 \quad (\ell = 2, p = 0.09) \\
10 &\rightarrow 110 \quad (\ell = 3, p = 0.09) \\
11 &\rightarrow 111 \quad (\ell = 3, p = 0.01)
\end{aligned}$$

  $L_2 = 0.81 \times 1 + 0.09 \times 2 + 0.09 \times 3 + 0.01 \times 3 = 1.29$ bits/block Per symbol: $L = L_2/2 = 0.645$ bits/symbol, $\eta = 0.469/0.645 \approx 72.7\%$

- **Block coding (n=3)**: Would get even closer to 0.469
- **Theoretical limit**: As $n \to \infty$, $L \to 0.469$

Key insight: The theorem tells us:

1. We can never beat 0.469 bits/symbol (impossibility)
2. We can get as close as we want to 0.469 bits/symbol (achievability)

Important

What the Theorem Does NOT Say

- **It doesn't say how to construct the code** - just that one exists
- **It doesn't guarantee practical implementation** - block size n might need to be huge
- **It doesn't account for computational complexity** - the code might be too complex to implement
- **It assumes we know the true probabilities** - in practice, we estimate them

Yet, this theorem is revolutionary because it:

1. Establishes a **fundamental limit** (like the speed of light in physics)
2. Provides a **benchmark** for evaluating compression algorithms
3. Guides algorithm design toward this limit

2.4.7 Key Takeaways

- Entropy measures both **average information** and **uncertainty**
- Higher-order models reduce entropy by exploiting dependencies
- The entropy rate represents the ultimate compression limit
- Shannon's theorem precisely separates the *possible* from the *impossible*

2.5 Entropy as a Lower Bound

2.5.1 The Fundamental Inequality

Theorem 2.2 (Entropy Lower Bound). *For any* ***uniquely decodable*** *code C for source X:*

$$L(C) \geq H(X)$$

where $L(C) = \mathbb{E}[\ell(X)] = \sum_i p_i \ell_i$ is the expected code length.

Example

Binary Source with $p(0) = 0.9$, $p(1) = 0.1$

$$H = -0.9 \log_2 0.9 - 0.1 \log_2 0.1 \approx 0.469 \text{ bits/symbol}$$

Why we can't achieve 0.4 bits/symbol:

1. For 100 symbols, typical sequences: $2^{100 \times 0.469} \approx 2^{46.9}$
2. To encode all uniquely, need at least $2^{46.9}$ codewords
3. At 0.4 bits/symbol, total bits = 40
4. # codewords $\leq 2^{40} < 2^{46.9} \to$ impossible!

2.6 Kraft-McMillan Inequality: Core Theoretical Tool

2.6.1 Statement and Interpretation

Theorem 2.3 (Kraft-McMillan Inequality (Binary Case)). *Let $\ell_1, \ell_2, \ldots, \ell_m$ be the lengths of codewords in a* ***prefix code****. Then:*

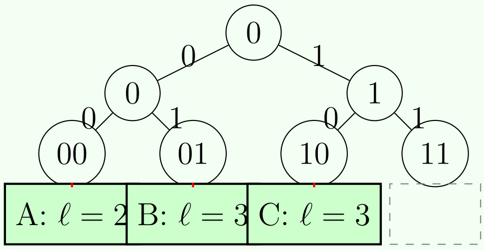
$$\sum_{i=1}^{m} 2^{-\ell_i} \leq 1$$

Conversely, if integers $\ell_1, \ldots, \ell_m$ satisfy this inequality, then there exists a binary prefix code with these lengths.

Example

Tree Visualization of Kraft Inequality

Consider a binary tree of depth $L = \max \ell_i$:



Calculating Kraft sum for $\{\ell_A = 2, \ell_B = 3, \ell_C = 3\}$:

$$\sum 2^{-\ell_i} = 2^{-2} + 2^{-3} + 2^{-3} = 0.25 + 0.125 + 0.125 = 0.5 \leq 1$$

Example

Testing Code Feasibility

1. **Valid lengths**: $\{1, 2, 3, 3\}$

$$\sum = 2^{-1} + 2^{-2} + 2^{-3} + 2^{-3} = 0.5 + 0.25 + 0.125 + 0.125 = 1.0 \quad \text{VALID}$$

2. **Invalid lengths**: $\{1, 1, 2\}$

$$\sum = 2^{-1} + 2^{-1} + 2^{-2} = 0.5 + 0.5 + 0.25 = 1.25 > 1 \quad \text{INVALID}$$

2.7 Optimality of Huffman Codes (Theory Only)

2.7.1 The Optimality Theorem

Theorem 2.4 (Huffman Optimality)**.** *Given a source with symbol probabilities* $p_1, p_2, \ldots, p_m$*, the Huffman algorithm produces a prefix code that* ***minimizes*** *the expected code length* $L = \sum_{i=1}^{m} p_i \ell_i$ *among all prefix codes.*

2.7.2 Relation to Entropy

For any Huffman code:

$$H(X) \leq L_{\text{Huffman}} < H(X) + 1$$

Example

Understanding the "+1" Gap

Consider source with probabilities $\{0.6, 0.3, 0.1\}$:

$$H \approx 1.295 \text{ bits}$$

Ideal (non-integer) lengths: $-\log_2 p_i = \{0.737, 1.737, 3.322\}$
Huffman code: 0.6→0, 0.3→10, 0.1→11

$$L = 0.6 \times 1 + 0.3 \times 2 + 0.1 \times 2 = 1.4 \text{ bits}$$

Comparison:

- Entropy: 1.295 bits
- Huffman: 1.400 bits
- Gap: 0.105 bits (much less than 1!)

2.7.3 Why Huffman is Optimal but Not Perfect

Important

Huffman is Optimal Within a Restricted Class
Huffman is optimal among:

- **Symbol-by-symbol** codes
- **Prefix** codes
- **Static** codes

But real optimality might require:

- **Block coding**
- **Fractional bits** (arithmetic coding)
- **Adaptive probabilities**

2.8 Limitations of Symbol-by-Symbol Coding

2.8.1 Three Fundamental Limitations

1. **Cannot Exploit Dependencies**
2. **Integer Length Constraint**: $\ell_i \in \mathbb{Z}^+$ but $-\log_2 p_i \in \mathbb{R}$
3. **Memoryless Assumption**

Example

English Text: The Cost of Symbol-by-Symbol

- **First-order entropy** (ignoring dependencies): 4.0 bits/letter
- **Actual entropy rate** (with dependencies): 1.5 bits/letter
- **Huffman on letters**: 4.0 bits/letter
- **Gap**: 2.5 bits/letter wasted due to ignoring dependencies

2.9 Block Coding and Improved Efficiency

2.9.1 The Block Coding Idea

Instead of coding symbols individually, group them into blocks of length n:

$$\mathbf{X} = (X_1, X_2, \ldots, X_n)$$

Definition

nth Extension of a Source
For a source with alphabet $\mathcal{X}$, the nth extension has alphabet:

$$\mathcal{X}^n = \{(x_1, \ldots, x_n) : x_i \in \mathcal{X}\}$$

with size $|\mathcal{X}|^n$.

2.9.2 Key Mathematical Results

Theorem 2.5 (Entropy of Block Source)**.** *For a discrete memoryless source:*

$$H(X^n) = nH(X)$$

Theorem 2.6 (Block Coding Performance)**.** *There exists a prefix code C_n for X^n such that:*

$$nH(X) \leq L_n < nH(X) + 1$$

Dividing by n:

$$H(X) \leq \frac{L_n}{n} < H(X) + \frac{1}{n}$$

Important

The Magic of Block Coding

As $n \to \infty$:

$$\frac{L_n}{n} \to H(X)$$

We can approach entropy **arbitrarily closely** by making blocks larger!

2.9.3 Step-by-Step Example

Example

Binary Source: $p(0) = 0.9$, $p(1) = 0.1$, $H \approx 0.469$

Step 1: $n = 1$ **(symbol-by-symbol)**

- Huffman: 0→0, 1→1
- $L_1 = 1$ bit/symbol
- Efficiency: $\eta = 0.469/1 = 46.9\%$

Step 2: $n = 2$ **(code pairs)**

- Block probabilities: $P(00) = 0.81$, $P(01) = 0.09$, $P(10) = 0.09$, $P(11) = 0.01$
- Codes: 00→0, 01→10, 10→110, 11→111
- Per symbol: $L_2/2 = 0.645$ bits/symbol
- Efficiency: $\eta = 0.469/0.645 = 72.7\%$

2.10 Trade-Offs in Block Coding

2.10.1 The Engineering Challenges

1. **Exponential Alphabet Growth**: $|\mathcal{X}^n| = |\mathcal{X}|^n$
2. **Memory Requirements**: Huffman tree has $2m^n - 1$ nodes
3. **Computational Complexity**: $O(m^n \log m^n)$
4. **Delay and Latency**: Must wait for n symbols

2.11 From Block Coding to Modern Compression

2.11.1 Arithmetic Coding: Fractional-Bit Block Coding

Important

Arithmetic Coding as "Infinite Block Coding"

Arithmetic coding cleverly avoids the exponential growth problem:

- **Idea**: Encode entire message as a single real number in [0,1)
- **No explicit blocks**: Processes symbols sequentially
- **Fractional bits**: Achieves $L \approx H(X)$ without large n
- **Removes integer constraint**: No "+1" overhead!

2.11.2 Context Modeling: Approximating Large Blocks

Instead of explicit block coding, modern compressors use:

1. **Context Models**: Predict next symbol based on previous k symbols
2. **Prediction + Residual Coding**: Encode only prediction error
3. **Dictionary Methods (LZ family)**: Build dictionary of previously seen phrases

2.12 What Theory Guarantees vs What Practice Achieves

Aspect	Theory Guarantees	Practice Achieves
Optimality	Can approach entropy arbitrarily closely	Gets close, but with practical limits
Block Size	$n \to \infty$ gives optimality	n limited by memory, latency, complexity
Complexity	Ignored, infinite resources allowed	Critical constraint; often dominates design

2.13 Summary and Key Takeaways

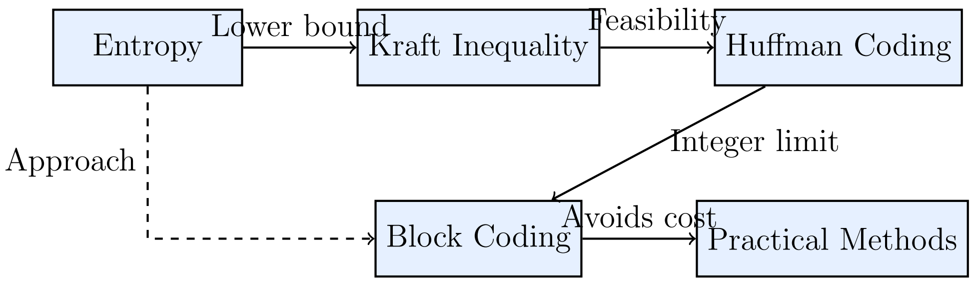
Important

Five Fundamental Lessons

1. **Entropy is the Absolute Limit**: $L \geq H(X)$ for any lossless code

2. **Kraft-McMillan Constrains All Codes**: $\sum 2^{-\ell_i} \leq 1$
3. **Huffman is Optimal Among Prefix Codes**: But limited by integer lengths
4. **Block Coding Allows Approaching Entropy**: $\lim_{n\to\infty} \frac{L_n}{n} = H(X)$
5. **Practical Compression Balances Efficiency and Complexity**

2.13.1 The Big Picture



2.13.2 Looking Forward

- **Next lecture: Arithmetic Coding**: Removes integer constraint
- **Then: Dictionary Methods (LZ family)**: Adaptive to data statistics
- **Finally: Modern Compressors**: Combining multiple techniques

Final Thought

Shannon's 1948 paper told us *exactly how good compression could possibly be.*
Every compressor since has been trying to approach that limit while staying within practical constraints.
The gap between theory and practice is where engineering creativity lives!

3: Lecture 3: Advanced Entropy Coding & Extensions

Lecture 3: Beyond Huffman – Advanced Entropy Coding Methods

3.1 Introduction & Motivation

Important

Recall Huffman Coding Limitations:

- **Integer code lengths**: Cannot reach entropy bound for highly skewed distributions
- **Static vs. Adaptive**: Standard Huffman requires prior knowledge of probabilities
- **Codebook overhead**: Need to transmit/store the coding tree
- **Symbol-by-symbol constraint**: Processes one symbol at a time

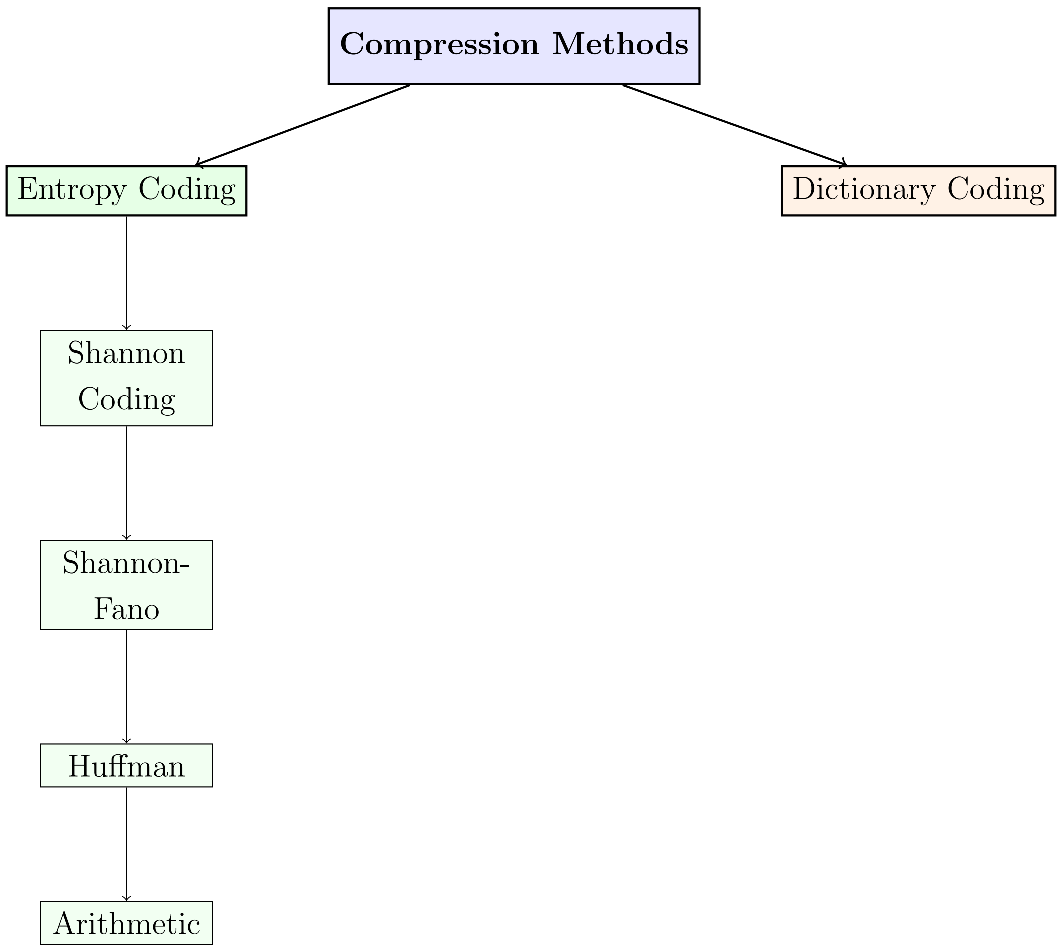
Lecture Roadmap:

1. **Framework**: Coding taxonomy and conceptual organization
2. **Historical methods**: Shannon & Shannon-Fano coding
3. **Practical improvements**: Canonical and Adaptive Huffman
4. **Next generation**: Arithmetic coding paradigm
5. **Synthesis**: Comparison and forward look

3.2 Coding Taxonomy & Framework

Definition

Coding Taxonomy: Classification of compression methods based on key characteristics



Key Dimensions in Compression Algorithm Design

1. **Modeling vs. Coding (Two-Stage View)**:
   - **Modeling Phase**: Analyzes data to estimate symbol probabilities or discover patterns.
     - *Examples*: Frequency counting (Huffman), context modeling (PPM), dictionary construction (LZ77)
   - **Coding Phase**: Converts modeled information into actual bits.
     - *Examples*: Huffman codes, arithmetic codes, LZ77 pointers
   - Some algorithms intertwine both (e.g., LZW builds dictionary while coding).
2. **Knowledge of Source Distribution**:
   - **Static/Fixed**: Uses a predefined model that doesn't change.
     - *Example*: JPEG Huffman tables, known language frequencies
     - Requires prior knowledge of data; fails if distribution differs.
   - **Adaptive**: Learns and updates the model during compression.
     - *Example*: Adaptive Huffman, LZ78 dictionary building

    - No prior knowledge needed; overhead for model transmission.
  - **Universal**: Can compress any source asymptotically optimally.
    - *Theoretical property*: LZ family, arithmetic with adaptive model
    - Note: Most adaptive methods are universal in practice.

3. **Processing Granularity**:
  - **Symbol-by-Symbol**: Each input symbol maps to one codeword.
    - *Example*: Huffman coding
    - Simple but limited to integer bits per symbol.
  - **Block Coding**: Fixed-size groups of symbols coded together.
    - *Example*: Block-sorting (BWT) processes blocks
    - Can capture inter-symbol dependencies within block.
  - **Stream/Incremental**: Continuous processing with immediate output.
    - *Example*: Arithmetic coding, LZ77 sliding window
    - No blocking delay; good for real-time applications.
  - *Note*: "Message-wide" (whole file as one symbol) is theoretical ideal; arithmetic coding approximates it by treating the entire stream as one long fractional code.

4. **Algorithmic Approach (Primary Taxonomy)**:
  - **Statistical Coding**: Uses probability estimates (Huffman, Arithmetic)
  - **Dictionary Coding**: Replaces repeated patterns with references (LZ family)
  - **Transform Coding**: Changes data domain then codes (DCT, wavelet)
  - **Predictive Coding**: Predicts next value, codes difference (DPCM, LPC)

Important Relationships:

- Adaptive methods are usually universal for practical purposes.
- Stream coding is possible with both symbol-by-symbol (Huffman) and message-wide approaches (arithmetic).
- Block processing (like BWT) is often followed by stream coding (like MTF+RLE+arithmetic in bzip2).
- Modeling and coding can be separated (PPM + arithmetic) or combined (LZW).

3.3 Shannon Coding (1948)

Definition

Shannon Coding: A constructive method derived from Shannon's source coding theorem that assigns codewords by taking the binary expansion of cumulative probabilities:

$$l_i = \lceil -\log_2 p_i \rceil \quad \text{and} \quad \text{code}_i = \text{First } l_i \text{ bits of } F_i$$

where p_i is the probability of symbol i, and $F_i = \sum_{j=1}^{i-1} p_j$ is the cumulative probability.

Example

Example: Given symbols with probabilities:

Symbol	Probability	$-\log_2 p_i$	Cumulative F_i
A	0.5	1.0	0.0
B	0.25	2.0	0.5
C	0.125	3.0	0.75
D	0.125	3.0	0.875

Step-by-step construction:

1. **Calculate lengths:** $l_A = \lceil 1.0 \rceil = 1$, $l_B = 2$, $l_C = 3$, $l_D = 3$
2. **Sort by probability** (already done above)
3. **Compute cumulative probabilities:**
   - $F_A = 0$ (first symbol)
   - $F_B = 0.5$ (just A's probability)
   - $F_C = 0.5 + 0.25 = 0.75$ (A + B)
   - $F_D = 0.5 + 0.25 + 0.125 = 0.875$ (A + B + C)
4. **Convert F_i to binary and take first l_i bits:**
   - **A:** $F_A = 0.0_{10}$ in binary is $0.0000\ldots_2$
     - Take first $l_A = 1$ bit: **0**
   - **B:** $F_B = 0.5_{10}$ in binary is $0.1000\ldots_2$
     - Take first $l_B = 2$ bits: **10**
   - **C:** $F_C = 0.75_{10}$ in binary is $0.1100\ldots_2$
     - Take first $l_C = 3$ bits: **110**
   - **D:** $F_D = 0.875_{10}$ in binary is $0.1110\ldots_2$

– Take first $l_D = 3$ bits: **111**

Resulting code:

Symbol	Probability	Shannon Code
A	0.5	0
B	0.25	10
C	0.125	110
D	0.125	111

Expected length: $L = 0.5 \times 1 + 0.25 \times 2 + 0.125 \times 3 + 0.125 \times 3 = 1.75$ bits/symbol

Important

Understanding the Binary Expansion Process:
When we write F_i in binary (e.g., $0.5 = 0.1_2$, $0.75 = 0.11_2$), we're essentially:

- Dividing the interval [0,1) into subintervals based on probabilities
- Each symbol gets an interval of size p_i
- The codeword is the **binary fraction** representing the **start** of that interval
- We use exactly $l_i = \lceil -\log_2 p_i \rceil$ bits, which ensures:

$$\frac{1}{2^{l_i}} \leq p_i < \frac{1}{2^{l_i - 1}}$$

- This guarantees unique prefixes because intervals don't overlap!

Important

Properties of Shannon Coding:

- **Constructive proof**: Demonstrates that prefix codes exist for any lengths satisfying Kraft inequality
- **Simple to compute**: Direct from probabilities, no tree needed
- **Not optimal**: Unlike Huffman, doesn't minimize expected length (compare: Huffman would give A=0, B=10, C=110, D=111 **same in this case!**)
- **Theoretical importance**: Foundation for Shannon's source coding theorem
- **Efficiency bound**: $H(X) \leq L < H(X) + 1$ (like Shannon's theorem says)

3.4 Shannon–Fano Coding (1949)

Definition

Shannon–Fano Coding: A top-down, recursive source coding technique that assigns binary codewords by repeatedly partitioning a set of symbols into two subsets whose total probabilities are as close as possible. The method was developed independently by *Claude Shannon* and *Robert Fano* in 1949.

Algorithm Description

High-level idea: Symbols with higher probabilities should receive shorter codewords. This is achieved by repeatedly splitting the symbol set into two probability-balanced groups and assigning binary prefixes.

Step-by-step procedure:

1. **Sort** the symbols in decreasing order of probability:

$$p_1 \geq p_2 \geq \cdots \geq p_n.$$

2. **Recursive partitioning:**

   - If the current set contains only one symbol, stop (this is the base case).
   - Find an index k that minimizes

$$\left| \sum_{i=1}^{k} p_i - \sum_{i=k+1}^{n} p_i \right|.$$

   - This divides the symbols into two subsets:

$$S_1 = \{1, \ldots, k\}, \qquad S_2 = \{k+1, \ldots, n\}.$$

   - Append bit **0** to the codewords of all symbols in S_1.
   - Append bit **1** to the codewords of all symbols in S_2.
   - Apply the same procedure recursively to S_1 and S_2.

Example with Six Symbols

Example

Example: Consider six symbols with the following probabilities.

Symbol	Probability	$-\log_2 p_i$
A	0.30	1.74
B	0.25	2.00
C	0.20	2.32
D	0.10	3.32
E	0.10	3.32
F	0.05	4.32

Construction process

Step 1: First split (balance 0.55 vs. 0.45)

- Sorted symbols: A(0.30), B(0.25), C(0.20), D(0.10), E(0.10), F(0.05)
- Best split: $\{A, B\}(0.55)$ and $\{C, D, E, F\}(0.45)$
- Prefix assignment: $\{A, B\} \to 0$, $\{C, D, E, F\} \to 1$

Step 2: Split $\{A, B\}$

- A: **00**, B: **01**

Step 3: Split $\{C, D, E, F\}$

- Best split: $\{C\}(0.20)$ and $\{D, E, F\}(0.25)$
- C: **10**, $\{D, E, F\} \to 11$

Step 4: Split $\{D, E, F\}$

- Best split: $\{D\}(0.10)$ and $\{E, F\}(0.15)$
- D: **110**, $\{E, F\} \to 111$

Step 5: Split $\{E, F\}$

- E: **1110**, F: **1111**

Final codes:

Symbol	Probability	Code	Length
A	0.30	00	2
B	0.25	01	2
C	0.20	10	2
D	0.10	110	3
E	0.10	1110	4
F	0.05	1111	4

Expected code length:

$$L = 2.40 \text{ bits/symbol.}$$

Entropy:

$$H(X) \approx 2.25 \text{ bits/symbol.}$$

Efficiency:

$$\frac{H(X)}{L} \approx 93.8\%.$$

Visual Tree Representation

```
                    ROOT
                   /    \
                 0/      \1
                 /        \
          {A,B} 0.55    {C,D,E,F} 0.45
             / \            / \
           0/   \1        0/   \1
           /     \        /     \
    {A} 0.30  {B} 0.25  {C} 0.20  {D,E,F} 0.25
      00        01        10          / \
                                    0/   \1
                                    /     \
                               {D} 0.10  {E,F} 0.15
                                 110         / \
                                           0/   \1
                                           /     \
                                     {E} 0.10  {F} 0.05
                                       1110      1111
```

Key Observations

- **Probability-balanced splits:** The algorithm focuses on balancing probabilities rather than the number of symbols.
- **Variable code lengths:** More probable symbols receive shorter codewords.
- **Prefix-free property:** No codeword is a prefix of another.
- **Near-optimal performance:** The efficiency is high but not guaranteed to be optimal.

Important

Limitations and Historical Context

- Shannon–Fano coding does *not* always produce the optimal code.
- Huffman coding (1952) guarantees the minimum average code length.

- Historically important as a precursor to Huffman coding.

3.5 Canonical Huffman Codes

Definition

Canonical Huffman Code: A standardized representation of a Huffman code in which:

- Codes are assigned in lexicographic (binary) order
- All codewords of the same length are consecutive binary numbers
- The first codeword of each length is the smallest possible binary value

Only the code lengths are required to reconstruct the entire code. This enables compact storage and fast table-based decoding.

Why Canonical Huffman Codes?

A standard Huffman algorithm produces *optimal code lengths*, but the exact bit patterns depend on implementation details such as tie-breaking and tree construction:

- Different trees can yield the same optimal set of code lengths
- Different bit assignments, but identical compression performance

Canonical Huffman coding fixes **one unique and deterministic assignment** for a given set of code lengths so that:

1. Only code lengths need to be transmitted
2. The decoder can reconstruct the codes without ambiguity
3. Decoding can be implemented efficiently using lookup tables

Two-Step Process

Complete workflow:

1. **Run the standard Huffman algorithm** to obtain optimal code lengths l_i
2. **Apply the canonical transformation** to convert lengths into standardized codes

Symbol probabilities $\xrightarrow{\text{Huffman}}$ Code lengths $\xrightarrow{\text{Canonical}}$ Canonical codes

Canonical Code Construction Algorithm

Input: Code lengths l_i obtained from a standard Huffman algorithm **Output:** Canonical Huffman codes

Algorithm 1 Canonical Huffman Construction

Require: Code lengths $l_1, \ldots, l_n$
Ensure: Canonical codes $c_1, \ldots, c_n$

Sort symbols by $(l_i, \text{symbol index})$
$l_{\min} \leftarrow \min\{l_i\}$, $l_{\max} \leftarrow \max\{l_i\}$
for $l = l_{\min}$ to $l_{\max}$ **do**
    $count[l] \leftarrow 0$ ▷ Initialize count array
end for
for each symbol i **do**
    $count[l_i] \leftarrow count[l_i] + 1$ ▷ Count symbols per length
end for
$next[l_{\min}] \leftarrow 0$
for $l = l_{\min} + 1$ to $l_{\max}$ **do**
    $next[l] \leftarrow (next[l-1] + count[l-1]) \times 2$
end for
for each symbol i (in sorted order) with length l_i **do**
    $c_i \leftarrow \text{binary}(next[l_i], l_i)$ ▷ Convert to l_i-bit binary
    $next[l_i] \leftarrow next[l_i] + 1$
end for

Detailed step-by-step explanation:

1. **Sort symbols:**
   (a) Primary key: increasing code length l_i
   (b) Secondary key: symbol value (or symbol index)
2. **Count symbols per length:** Let $L_{\min}$ and $L_{\max}$ be the minimum and maximum code lengths. For each length l, compute:

$$\text{count}[l] = \text{number of symbols with length } l$$

3. **Compute starting codes:** Initialize the first starting code:

$$\text{start\_code}[L_{\min}] = 0$$

   For each length $l = L_{\min} + 1$ to $L_{\max}$:

$$\text{start\_code}[l] = \big(\text{start\_code}[l-1] + \text{count}[l-1]\big) \times 2$$

4. **Initialize assignment pointers:**

$$\text{next\_code}[l] = \text{start\_code}[l] \quad \text{for all } l$$

5. **Assign codes to symbols:**

   - Traverse the sorted symbol list
   - For each symbol of length l:
     - Assign the current value of next_code[l]
     - Increment next_code[l] by one

Understanding the Shift Operation

Key idea: All codes of length $l+1$ must begin immediately after the last code of length l, and must be exactly one bit longer.

If the last code of length l has integer value x, then the first code of length $l+1$ is:

$$\text{start\_code}[l+1] = (x+1) \times 2$$

In binary representation, this corresponds to:

$$(x+1)_2 \text{ followed by a } 0$$

This construction ensures:

- Prefix-free property
- Lexicographic ordering is preserved
- No overlap between different length groups

Complete Worked Example

Example

Example: Suppose the Huffman algorithm produces the following code lengths:

Symbol	Length (l_i)
A	2
B	3
C	3
D	3
E	4
F	4

Step 1: Sort symbols by length, then by symbol order

- Length 2: A
- Length 3: B, C, D
- Length 4: E, F

Step 2: Count symbols per length

$$\text{count}[2] = 1, \qquad \text{count}[3] = 3, \qquad \text{count}[4] = 2$$

Step 3: Compute starting codes

$$\begin{aligned}
&\text{start_code}[2] = 0 \quad (00_2)\\
&\text{start_code}[3] = (0 + 1) \times 2 = 2 \quad (010_2)\\
&\text{start_code}[4] = (2 + 3) \times 2 = 10 \quad (1010_2)
\end{aligned}$$

Step 4: Initialize assignment pointers

$$\text{next\_code}[2] = 0, \quad \text{next\_code}[3] = 2, \quad \text{next\_code}[4] = 10$$

Step 5: Assign codes

Symbol	Length	Action	Code	next_code[l] after
A	2	Assign next_code[2] = 0	00	1
B	3	Assign next_code[3] = 2	010	3
C	3	Assign next_code[3] = 3	011	4
D	3	Assign next_code[3] = 4	100	5
E	4	Assign next_code[4] = 10	1010	11
F	4	Assign next_code[4] = 11	1011	12

Final canonical codes:

Symbol	Length	Canonical Code
A	2	00
B	3	010
C	3	011
D	3	100
E	4	1010
F	4	1011

Transmission and Decoding

Transmission format:

- Transmit only the sequence of code lengths:

$$\langle l_1, l_2, \ldots, l_n \rangle$$

- Example: $\langle 2, 3, 3, 3, 4, 4 \rangle$
- Each length can be represented using $\lceil \log_2 L_{\max} \rceil$ bits

Decoder operation:

1. Read code lengths for all symbols
2. Reconstruct canonical codes using the same algorithm
3. Build a decoding table mapping $(\text{code}, \text{length})$ to symbols
4. Decode the bitstream using table lookup

Comparison with Standard Huffman

Standard Huffman (tree-based)	**Canonical Huffman (length-based)**
Output: Huffman tree	Output: Code lengths
Must transmit tree structure	Transmit only lengths
Decoding by tree traversal	Decoding by table lookup
Multiple equivalent trees	Single deterministic assignment
Same optimal compression	Same optimal compression
More complex implementation	Simple, robust implementation

Key Properties

- **Optimality:** Identical compression ratio to standard Huffman
- **Compactness:** Only code lengths are stored or transmitted
- **Speed:** Table-based decoding is significantly faster
- **Standardization:** Used in DEFLATE (gzip/ZIP), JPEG, PNG, MPEG

Important

Critical Insight: Canonical Huffman coding is **not** a different compression algorithm. The Huffman algorithm determines *how long* each code should be, while the canonical transformation determines the *exact bit patterns* in a consistent and reproducible manner.

3.6 Adaptive Huffman Coding

3.6.1 Overview

Definition

Adaptive Huffman Coding is a single-pass, lossless data compression technique in which the Huffman model is updated dynamically as symbols are encoded and decoded. The Huffman tree evolves on-the-fly, allowing both the encoder and decoder to adapt to changing symbol statistics without any prior knowledge of the source distribution.

Unlike static Huffman coding, adaptive Huffman coding does not require a preprocessing phase to compute symbol frequencies. Instead, symbol statistics are learned incrementally as the data stream is processed.

3.6.2 Limitations of Static Huffman Coding

Static Huffman coding assumes that symbol statistics are known in advance:

- Requires two passes over the data:
  1. Collect symbol frequencies
  2. Encode using the constructed Huffman tree
- The Huffman tree (or code lengths) must be transmitted as side information
- Cannot adapt to non-stationary or evolving symbol distributions

These limitations motivate adaptive schemes when symbol statistics are unknown or change over time.

3.6.3 Key Principle of Adaptive Huffman Coding

Adaptive Huffman coding maintains a Huffman tree that is updated after encoding or decoding each symbol.

Key invariant: After processing each symbol, the encoder and decoder maintain *identical Huffman trees* by applying the same updates in the same order. This ensures correct decoding without transmitting the tree explicitly.

3.6.4 Initialization

The algorithm begins with a special symbol:

- A single **NYT** (Not Yet Transmitted) node

When a symbol appears for the first time:

- The code for the NYT node is output
- The raw symbol value is transmitted
- The NYT node is expanded into:
  - A new NYT node
  - A leaf node representing the new symbol

Note: Some variants preinitialize the tree if the alphabet is fixed, but the standard adaptive Huffman algorithm begins with only the NYT node.

3.6.5 Tree Update Mechanism

After each symbol is processed:

- The frequency (weight) of the corresponding leaf node is incremented
- The tree is updated to preserve the **sibling property**

Definition

Sibling Property: Nodes are numbered such that nodes with higher weights have higher numbers. For any given weight, all nodes with that weight form a contiguous block. Each internal node has exactly two children.

To restore this property, nodes may be swapped with others of equal weight, followed by incrementing parent node weights. This update process proceeds bottom-up toward the root.

3.6.6 Encoding and Decoding Process

- If the symbol has appeared before:
  - Output its current Huffman code
- If the symbol is new:
  - Output the code for the NYT node
  - Output the symbol in raw (fixed-length) form
- Update the tree identically at both encoder and decoder

Note: Exact bit patterns depend on the update order and implementation. Adaptive Huffman coding guarantees prefix-free optimality but not unique codes.

3.6.7 Algorithms

Two classical adaptive Huffman algorithms are widely used:

- **FGK Algorithm** (Faller–Gallager–Knuth)
- **Vitter's Algorithm (Algorithm V)**, which improves worst-case performance and is commonly preferred

Both algorithms maintain the sibling property while ensuring efficient updates.

3.6.8 Performance Characteristics

- Update cost is **amortized** $O(1)$ per symbol
- Worst-case update time is proportional to the tree height
- Compression efficiency:
  - Initially suboptimal
  - Converges toward entropy-optimal coding as statistics stabilize

3.6.9 Comparison with Static Huffman Coding

Feature	Static Huffman	Adaptive Huffman
Number of passes	Two	One
Prior statistics required	Yes	No
Model transmission	Required	Not required
Adaptation to changes	No	Yes
Initial efficiency	High	Low
Asymptotic efficiency	Optimal	Near-optimal

3.6.10 Applications

Adaptive Huffman coding is well suited for:

- Streaming data sources
- Real-time communication
- Situations where full statistics cannot be collected in advance

However, it may be less suitable when symbol distributions are known and stable, or when computational simplicity is critical.

3.6.11 Summary

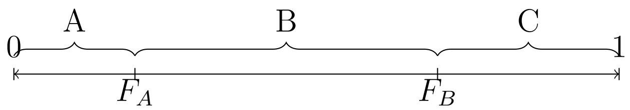
Adaptive Huffman coding extends classical Huffman coding to dynamic environments. By updating the coding model incrementally and maintaining strict structural invariants, it enables efficient, single-pass compression without prior knowledge of source statistics.

3.7 Arithmetic Coding: The Paradigm Shift

Definition

Arithmetic Coding: Encodes an entire message into a single fractional number in the interval $[0, 1)$, approaching the entropy bound very closely.

Core Idea: Represent messages as subintervals of $[0, 1)$:



Algorithm 2 Arithmetic Encoding Algorithm

Require: Message $m = s_1 s_2 \dots s_k$, symbol probabilities p_i
Ensure: Final interval $[low, high)$

$low \leftarrow 0.0$, $high \leftarrow 1.0$
for each symbol s in m **do**
  $range \leftarrow high - low$
  $high \leftarrow low + range \times F_s$ ▷ F_s: cumulative prob up to s
  $low \leftarrow low + range \times F_{s-1}$ ▷ F_{s-1}: cumulative prob before s
end forreturn any number in $[low, high)$

Example

Example: Encode message "CAB" with probabilities: A(0.5), B(0.25), C(0.25)

Symbol	Probability	Cumulative
A	0.5	0.5
B	0.25	0.75
C	0.25	1.0

Encoding:

1. Start: $[0, 1)$
2. Process 'C': $[0.75, 1.0)$ (C occupies $[0.75, 1.0)$)
3. Process 'A': $range = 0.25$
   - $low = 0.75 + 0.25 \times 0.0 = 0.75$

- $high = 0.75 + 0.25 \times 0.5 = 0.875$
- New interval: $[0.75, 0.875)$

4. Process 'B': $range = 0.125$
   - $low = 0.75 + 0.125 \times 0.5 = 0.8125$
   - $high = 0.75 + 0.125 \times 0.75 = 0.84375$
   - Final interval: $[0.8125, 0.84375)$

Output: Any number in $[0.8125, 0.84375)$, e.g., 0.8125 in binary

Important

Practical Implementation Issues:

- **Finite precision**: Use integer arithmetic with scaling
- **Renormalization**: Output bits when interval confined to one half
- **Carry-over**: Handle when interval spans midpoint
- **Termination**: Need special end-of-message symbol

Adaptive Arithmetic Coding: Easier than adaptive Huffman - just update probabilities as you go!

3.8 Comparison & Synthesis

Method	Optimal?	Adaptive?	Complexity	Near Entropy?	Key Applications
Shannon Coding	No	No	Low	No	Theoretical proofs
Shannon-Fano	No	No	Low	Sometimes	Historical
Huffman	Yes*	No	Low	Moderate	General purpose
Canonical Huffman	Yes*	No	Low	Moderate	DEFLATE, JPEG, PNG
Adaptive Huffman	Yes*	Yes	Medium	Moderate	Early compressors
Arithmetic Coding	**Near-opt**	**Yes**	**High**	**Yes (close)**	**JPEG2000, H.264, HEVC**

*Optimal for symbol-by-symbol coding given probabilities

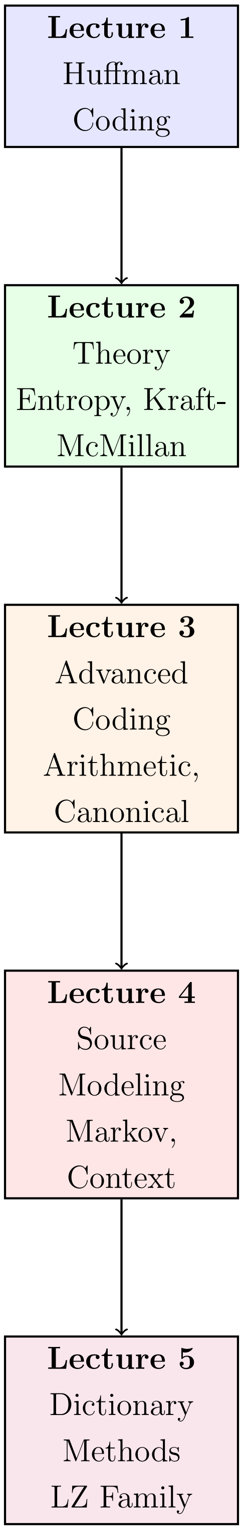
Important

Key Insights:

- **Huffman vs. Arithmetic**: Huffman is simpler but has an "integer penalty"; Arithmetic approaches entropy bound
- **Modern standards**: Arithmetic coding (CABAC) used in video compression for 10-20% better compression
- **Practical choice**: For general compression, Canonical Huffman (DEFLATE); for media, Arithmetic coding
- **The missing piece**: All these methods assume we have good probability estimates. Where do those come from?

3.9 Forward Look

The Complete Picture: What Comes Next?



Next Lecture: Source Modeling and Statistical Dependence

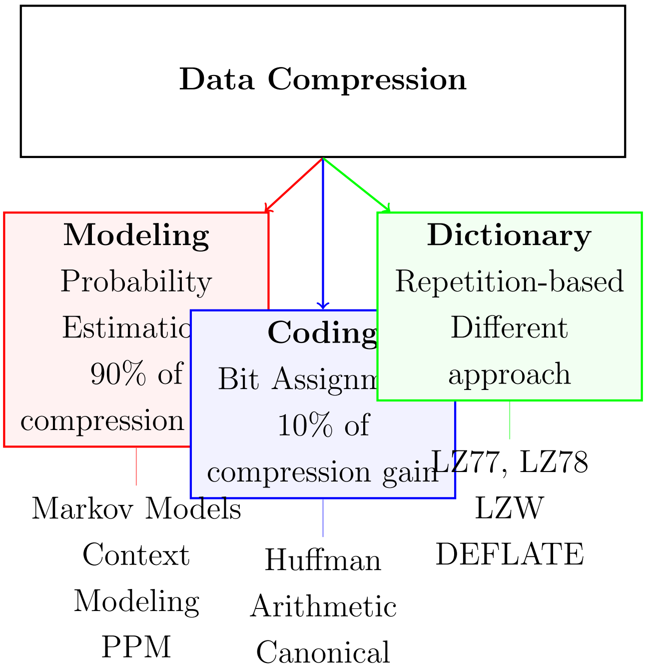
- **The missing half**: We now know how to code efficiently, but where do the probabilities come from?
- **Real data has memory**: 'Q' is usually followed by 'U' in English text
- **Markov models**: Capturing dependencies between symbols

- **Context modeling**: Using past symbols to predict future ones
- **The modeling-coding separation**: Modern compressors separate these two tasks

The Big Question for Next Time:

If Arithmetic coding can get within 0.01 bits of entropy, what's the real limit to compression?
The answer: It's not the coding, it's the *modeling*!

The Two Pillars of Compression (Revised View):



Exercise 3.0

Exercise 3.1: Given symbols with probabilities: A(0.4), B(0.3), C(0.2), D(0.1)

(a) Construct a Shannon code and compute its expected length

(b) Construct a Shannon-Fano code

(c) Compare with Huffman code from Lecture 2

Exercise 3.1

Exercise 3.2: Convert the following Huffman code to canonical form:

Symbol	Huffman Code
A	0
B	100
C	101
D	110
E	1110
F	1111

Exercise 3.2

Exercise 3.3: Encode the message "ABAC" using arithmetic coding with probabilities: A(0.6), B(0.3), C(0.1). Show each step.

Exercise 3.3

Exercise 3.4: Thinking Ahead: Consider the English phrase "THE QUICK BROWN FOX"

(a) If we use Huffman coding with letter frequencies, what's wrong with this approach?

(b) How might knowing that 'Q' is usually followed by 'U' help compression?

(c) Why would arithmetic coding be better than Huffman for this kind of data?

End of Lecture 3 – Advanced Entropy Coding Methods

Next: Lecture 4 – Source Modeling and Statistical Dependence

We now have efficient coding methods. Next: Where do the probabilities come from?

4: Lecture 4: Source Modeling and Statistical Dependence

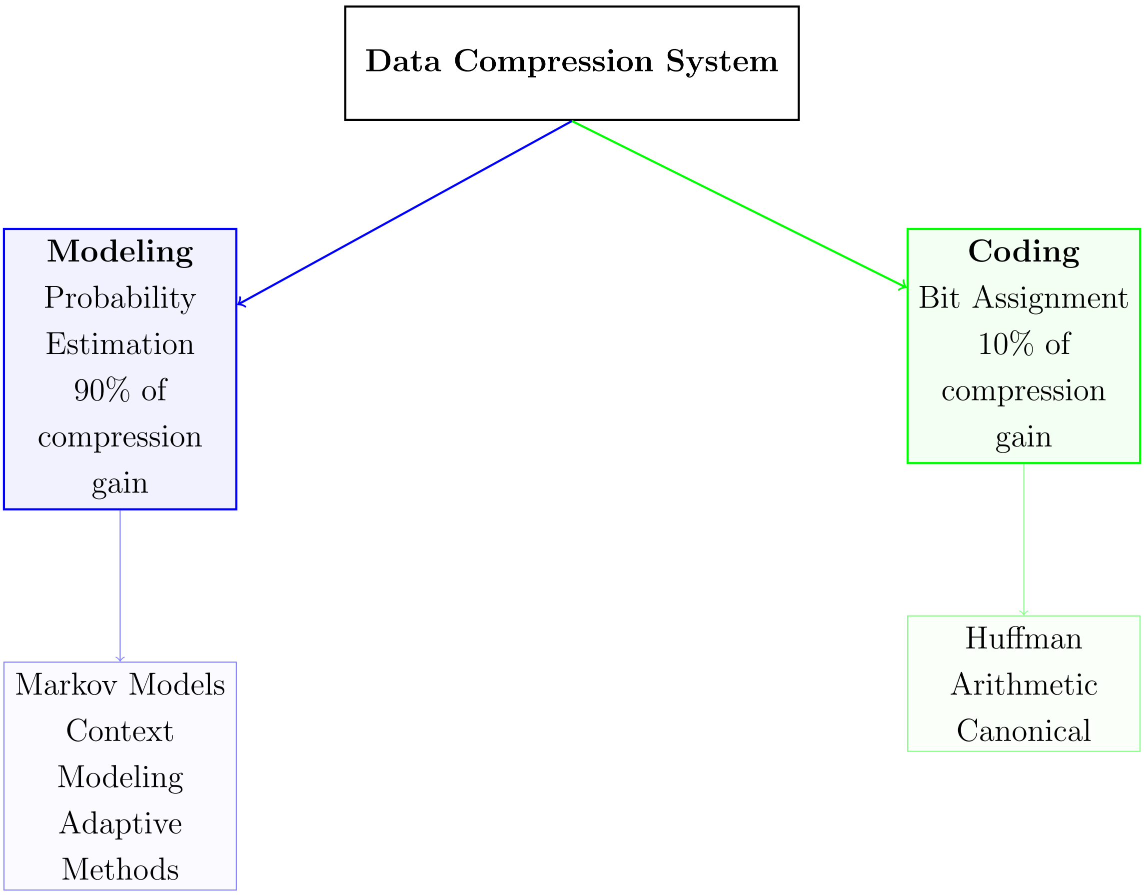
Lecture 4: Beyond Coding – The Power of Source Modeling

4.1 Introduction: Beyond Coding

Important

Key Insight: In the first three lectures, we've focused on **coding** - efficient ways to represent symbols given their probabilities. Now we address the other half: **modeling** - how to get good probability estimates in the first place.

The Big Picture:



Why Real Data Defies IID Assumptions:

- **IID (Independent Identically Distributed)**: Assumption behind simple Huffman
- **Reality**: Data has **memory** and **dependencies**
- Example: In English text, 'Q' is almost always followed by 'U'

- Example: In images, neighboring pixels are highly correlated

Today's Roadmap:

1. Understand statistical dependence in data
2. Learn Markov models for capturing memory
3. Explore context modeling techniques
4. See practical examples with real data
5. Connect modeling to coding (Lecture 3)

4.2 Memoryless vs. Sources with Memory

Definition

Memoryless Source (IID): Each symbol is generated independently of all previous symbols. Probability distribution: $P(X_n = x) = p(x)$ for all n.

Definition

Source with Memory: The probability of a symbol depends on previous symbols. Example: $P(X_n = x | X_{n-1} = y, X_{n-2} = z, \dots)$.

Example

Examples of Dependence in Real Data:

- **Text**: 'TH' is common, 'TQ' is rare
- **Images**: Neighboring pixels have similar colors
- **Audio**: Sound waves have temporal continuity
- **Video**: Consecutive frames are nearly identical
- **Source code**: Keywords, variable names repeat

Important

Measuring Dependence: Autocorrelation

$$\rho(k) = \frac{\mathbb{E}[(X_t - \mu)(X_{t+k} - \mu)]}{\sigma^2}$$

where k is the lag. High $\rho(k)$ means strong dependence at distance k.

4.3 Conditional Entropy and Mutual Information

Definition

Conditional Entropy: The average uncertainty about X given knowledge of Y:

$$H(X|Y) = -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\log_2 p(x|y)$$

Example

Intuition: "Knowing the Past Helps Predict the Future"
Consider English letters:

- Unconditional: $H(\text{letter}) \approx 4.07$ bits
- Given previous letter: $H(\text{letter}|\text{previous}) \approx 3.36$ bits
- Given previous 2 letters: $H(\text{letter}|\text{previous 2}) \approx 2.77$ bits

Each additional letter of context reduces uncertainty!

Definition

Mutual Information: Measures how much knowing Y reduces uncertainty about X:

$$I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)$$

Example

Worked Example: English Letter Dependence
Let X = current letter, Y = previous letter.

$$\begin{aligned} H(X) &= 4.07 \text{ bits} \\ H(X|Y) &= 3.36 \text{ bits} \\ I(X;Y) &= 4.07 - 3.36 = 0.71 \text{ bits} \end{aligned}$$

This means knowing the previous letter gives us 0.71 bits of information about the current letter.

4.4 Markov Sources

Definition

Markov Property (Memory-m): The future depends only on the last m symbols:

$$P(X_n = x|X_{n-1}, X_{n-2}, \ldots, X_1) = P(X_n = x|X_{n-1}, \ldots, X_{n-m})$$

First-Order Markov Model ($m = 1$):

- Only the immediately previous symbol matters
- Represented by transition probabilities $p_{ij} = P(X_n = j|X_{n-1} = i)$
- Can be shown as a state diagram or transition matrix

Example

Example: Weather Prediction Markov Chain
States: $\{\text{Sunny (S)}, \text{Rainy (R)}\}$
Transition probabilities:

	S	R
S	0.8	0.2
R	0.3	0.7

Interpretation: If today is sunny, 80% chance tomorrow is sunny, 20% chance rainy.

Higher-Order Markov Models:

- Order-k: Depends on last k symbols
- More accurate but exponentially more parameters
- Number of parameters grows as $|\mathcal{A}|^{k+1}$ where $\mathcal{A}$ is alphabet size

Important

Memory-Complexity Trade-off:

- **Order 0**: 26 parameters for English (simple but weak)
- **Order 1**: $26 \times 26 = 676$ parameters
- **Order 2**: $26 \times 26 \times 26 = 17,576$ parameters
- **Order 5**: $26^6 \approx 308$ million parameters!

This is the **context explosion problem**.

4.5 Entropy Rate Revisited

Definition

Entropy Rate of a Stationary Source:

$$H(\mathcal{X}) = \lim_{n\to\infty} \frac{1}{n} H(X_1, X_2, \ldots, X_n)$$

For a stationary Markov chain of order m:

$$H(\mathcal{X}) = H(X_{m+1}|X_1, \ldots, X_m)$$

Example

Calculating Entropy Rate for First-Order Markov Chain:
For weather example with stationary distribution $\pi = [\pi_S, \pi_R]$:

$$H(\mathcal{X}) = \pi_S H(X|S) + \pi_R H(X|R)$$

where:

$$\begin{aligned}
H(X|S) &= -0.8\log_2 0.8 - 0.2\log_2 0.2 \approx 0.7219 \\
H(X|R) &= -0.3\log_2 0.3 - 0.7\log_2 0.7 \approx 0.8813 \\
\pi_S &= 0.6, \quad \pi_R = 0.4 \quad \text{(solve } \pi P = \pi\text{)} \\
H(\mathcal{X}) &= 0.6 \times 0.7219 + 0.4 \times 0.8813 \approx 0.788 \text{ bits}
\end{aligned}$$

Important

What This Means for Compression:

- **IID assumption**: Limit $= H(X)$ (e.g., 4.07 bits/letter for English)
- **With modeling**: Limit $= H(\mathcal{X})$ (e.g., 2.3 bits/letter for English)
- **Potential gain**: Up to 45% better compression!

4.6 Context Modeling in Practice

Definition

Context Modeling: Maintain separate probability distributions for each possible context (history).

Fixed-Length vs. Variable-Length Contexts:

- **Fixed-length**: Always use last k symbols as context

- **Variable-length**: Use longest matching context in database
- Example: PPM (Prediction by Partial Matching) uses variable-length

Example

The Context Explosion Problem:

For English text (26 letters + space):

Order	Contexts	Parameters
0	1	27
1	27	729
2	729	19,683
3	19,683	531,441
4	531,441	14,348,907
5	14,348,907	387,420,489

By order 5: 387 million parameters need estimation!

Solutions to Context Explosion:

1. **Escaping**: Fall back to lower-order model when context unseen
2. **Blending**: Combine predictions from different order models
3. **Pruning**: Remove low-frequency contexts
4. **Adaptive methods**: Update probabilities as data arrives

4.7 Case Study: Text Compression Modeling

Example

English Text Compression with Different Models:

Model Type	Bits/Letter	Compression vs. ASCII
ASCII (baseline)	8.00	0%
Order-0 (Huffman)	4.07	49%
Order-1 (Bigram)	3.36	58%
Order-2 (Trigram)	2.77	65%
Order-3	2.43	70%
Order-5 (PPM)	2.23	72%
Optimal (Shannon 1951)	1.3	84%

Note: Each step improves compression by better modeling!

PPM (Prediction by Partial Matching):

- Uses **variable-length** contexts

- Tries highest-order model first
- Escapes to lower order if context unseen
- Blends probabilities from different orders
- State-of-the-art for text compression in 1990s

Important

Practical Entropy Reduction:

$$\begin{aligned}\text{IID model (Huffman)} &: 4.07 \text{ bits/letter}\\ \text{With context modeling} &: 2.23 \text{ bits/letter}\\ \text{Improvement} &: 45\% \text{ better compression!}\end{aligned}$$

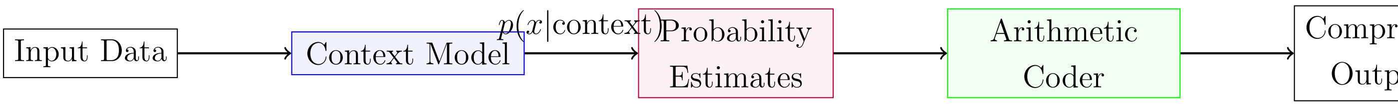
4.8 The Modeling–Coding Separation Principle

Definition

Modeling–Coding Separation: Modern compressors separate probability estimation (modeling) from bit assignment (coding).

Historical Evolution:

- **Early**: Integrated (Huffman builds tree from frequencies)
- **Modern**: Separated (Model → Probabilities → Arithmetic Coder)



Example

How PPM + Arithmetic Beats Huffman:

1. **PPM**: Sees context "TH" → predicts E with 80% probability
2. **Arithmetic**: Encodes E using $p = 0.8 \to$ 0.32 bits
3. **Huffman**: Would need at least 1 bit for any symbol
4. **Gain**: 0.32 bits vs 1+ bits = 3× better for this symbol!

Important

Why Arithmetic Coding is the Perfect Backend:

- Can handle **fractional bits** per symbol
- Accepts **changing probabilities** symbol by symbol
- Works with **adaptive models** naturally
- Achieves entropy bound for good models

4.9 Adaptive vs. Static Modeling

Definition

Static Models: Train once on representative data, use fixed model for all files.

- **Pros**: Fast encoding/decoding
- **Cons**: Model may not match specific file
- **Example**: Early text compressors using English statistics

Definition

Semi-Adaptive (Two-Pass): First pass: collect statistics; Second pass: encode.

- **Pros**: Tailored to specific file
- **Cons**: Need to transmit model (overhead)
- **Example**: Standard Huffman with tree transmission

Definition

Fully Adaptive (One-Pass): Update model while encoding/decoding.

- **Pros**: No model transmission, adapts to local changes
- **Cons**: Slower, initial poor compression
- **Example**: Adaptive Huffman, PPM with update

Example

Comparison in Practice:

Type	Compression	Speed	Memory
Static	Medium	Fast	Low
Semi-Adaptive	Good	Medium	Medium
Fully Adaptive	Best	Slow	High

Choice depends on application constraints!

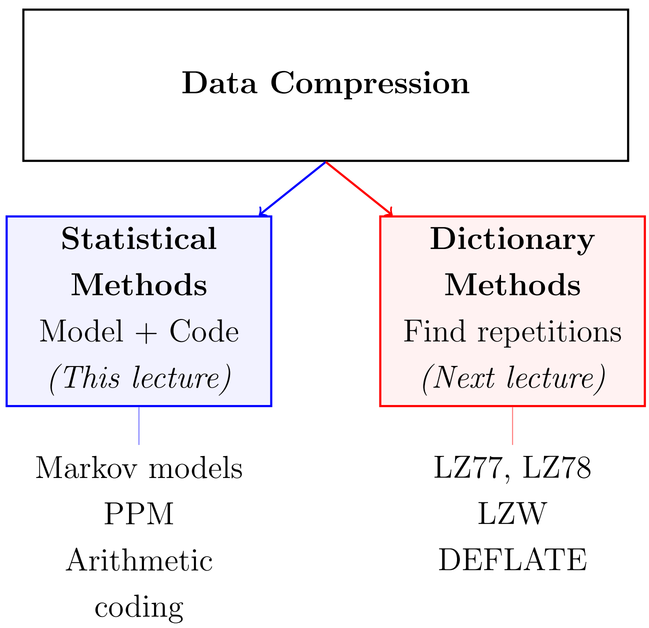
4.10 Summary and Forward Look

Important

Key Takeaways:

1. **The real compression is in modeling**, not just coding
2. **Context matters**: Using past symbols reduces uncertainty
3. **Markov models** capture memory in data
4. **Context explosion** limits practical model order
5. **Arithmetic coding** enables efficient use of good models
6. **Adaptive methods** avoid model transmission overhead

The Two Pillars Revisited:



Preview: Next Lecture on LZ Family (Dictionary Methods):

- A completely different approach: find repeating patterns

- No probability estimation needed
- Works well for files with exact repetitions
- Used in ZIP, GIF, PDF, and many modern formats
- Often combined with statistical methods in practice

Exercise 4.0

Exercise 4.1: Given the first-order Markov chain for weather:

	S	R
S	0.7	0.3
R	0.4	0.6

a) Find the stationary distribution $\pi = [\pi_S, \pi_R]$ b) Calculate the entropy rate $H(\mathcal{X})$ c) How does this compare to an IID source with $P(S) = 0.55, P(R) = 0.45$?

Exercise 4.1

Exercise 4.2: Consider the text fragment: "THE CAT SAT ON THE MAT" a) Build an order-1 (bigram) model for this text b) Calculate $H(X)$ (order-0 entropy) c) Calculate $H(X|Y)$ where Y is previous letter (order-1 conditional entropy) d) How much mutual information exists between consecutive letters?

Exercise 4.2

Exercise 4.3: Explain why arithmetic coding is better than Huffman coding when used with: a) A high-order Markov model b) An adaptive context model c) A model that gives very skewed probabilities (e.g., $p = 0.99$ for one symbol)

End of Lecture 4 – Source Modeling and Statistical Dependence

Next: Dictionary-based Compression (LZ Family)